PERFORMANCE OVERHEAD WHILE USING THREAD SYNCHRONIZATION TECHNIQUES

Author: Dumitru CALANCEA
Scientific adviser: senior lecturer Dumitru CIORBĂ

Technical University of Moldova
Email: deeemaaa@gmail.com

Abstract: *Thread synchronization mechanisms are widely used in software development for assuring the correct work of multithreaded applications, especially the correct access to shared variables. However these mechanisms come at a price and this price is the performance of the system. This article presents different ways in which thread synchronization techniques can cause performance drops. Also there are presented various ways of avoiding these situations by choosing alternative ways of designing the system and using appropriate data structures and mechanisms.*

Keywords: *threading, concurrency, synchronization, performance, optimization.*

1. The use of thread synchronization mechanisms

Multithreaded applications are widely used in today's software applications because they provide benefits such as improved system responsiveness, handling more parallel requests, etc. However in order to avoid simultaneous access to the same shared resource, the application should use some synchronization mechanisms. Otherwise the simultaneous action of more threads to the same resource can result in data corruption and other unwanted effects.

Modern programming languages provide the solution to such problems: synchronization mechanisms. Most common ones are locks, mutexes, and synchronization events. Because these mechanisms are pretty easy to use and provide great benefits to the reliability of the system they became widely used in multithreaded applications.

But these mechanisms don't come for free. They cause performance overhead from several reasons. One of the reasons is that in a poorly designed system one thread can spend too much time inside a critical section locking therefore the other threads. Another case is that intensive use of synchronization mechanisms for small/simple critical sections will cause a performance overhead because of the expensive synchronization mechanism used.

2. Too long lock time

By using an improperly designed critical section, the thread can end up spending too much time inside the critical section or to fail to reset the locking mechanisms in order to let other threads. This is bad because the other threads that need to access the critical resource will get stuck waiting for access. It can happen in the case of blocking two critical resources at the same time. While the thread inside the critical section will use just a resource at a time, the other one will still remain unavailable.

Deadlock is also a serious problem one can encounter when working with threads. Deadlock occurs when two or more tasks are waiting for each other to release a lock.[1] So because both resources can't be blocked simultaneously there is a chance that another thread will block one of the resources just after the first one blocked its first resource. Deadlock can cause system stall.

3. Performance overhead of the synchronization mechanism itself

Although at the first glance the performance overhead of the blocking/unblocking process can seem insignificant compared to the advantages and safety the synchronization mechanism provides, at a closer look it can turn out be the vice-versa. This is especially valid when the critical section contains fast

operations like addition or subtraction to a shared variable, especially if the critical section is called inside a loop.

The problem is that quite often the process of blocking/unblocking process is not very fast. There are differences not only between unsynchronized and synchronized algorithms, but between different synchronization mechanisms as well. It takes almost 100 times longer to lock an unowned mutex than it does to lock an unowned critical section because the critical section can be done in user mode without involving the kernel. [2]

Even the simplest and fastest synchronization mechanisms, like the lock consist of some comparisons, additions, and assignments. Although it takes some time to lock an unowned lock, in a Windows system the lock is still way faster than a mutex. This happens because locks work in user mode while mutexes work in kernel mode, and the context switch is by itself an expensive operation.

An experiment was made in order to have an approximate view on the overhead of some synchronization techniques. A loop of one million increments-decrements was run in a single-threaded environment 3 times in C#. First time no synchronization technique was used. Second time .NET locks were used, and the third time a .NET mutex was used. Although the results presented in Table 1 should not be taken as a benchmark, they give an idea about how big the performance overhead for each synchronization technique is. The time for the no synchronization run was taken as reference.

Table 1

No Synchronization	Lock	Mutex
1	5.4	362

The most basic synchronization primitive is the lock and it's the most efficient in memory use and execution time. The mutex uses more memory than a lock. [3]

4. Minimizing overheads

First thing to think about when trying to optimize a multithreaded code is weather the given piece of code needs or doesn't need to be synchronized. Read-only objects for example don't need to be synchronized. Some objects can implement a copy-on-write mechanism, so they also don't need to be synchronized.

Selecting the appropriate synchronization technique is also a good idea. Avoiding using slow synchronization mechanisms when faster ones are perfectly valid for the given task can boost up the performance of the whole program.

Sometimes developers design a class to be thread-safe just because it's impossible at the time to anticipate weather it will be used by multiple threads or not. This is also a point that should be checked.

When some bottleneck that uses synchronized objects is detected, sometimes the synchronization can be removed by replacing the synchronized object with local unsynchronized copies.

Also synchronized wrappers can be designed over unsynchronized classes in order to assure the possibility of further declaration of synchronized as well as unsynchronized objects of the class.

When no sharing is intended, give each thread a private copy of the data. When sharing is important, provide explicit synchronization to make certain that the program behaves in a deterministic manner. [4]

Bibliography

1. Colin Campbell, Ade Miller, *Parallel Programming with Microsoft Visual C++*. Microsoft, 2011, p.5.
2. Beveridge J., Wiener R. *Multithreading Applications in Win32.* Addison-Wesley Developers Press, 1996, p.85.
3. Fayez Gebali. *Algorithms and Parallel Computing.* University of Victoria, Victoria, 2011, p.81.
4. Sun Microsystems. *Multithreaded Programming Guide*. Sun Microsystems, 2008, p.213